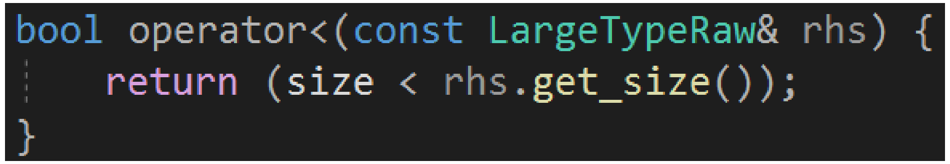
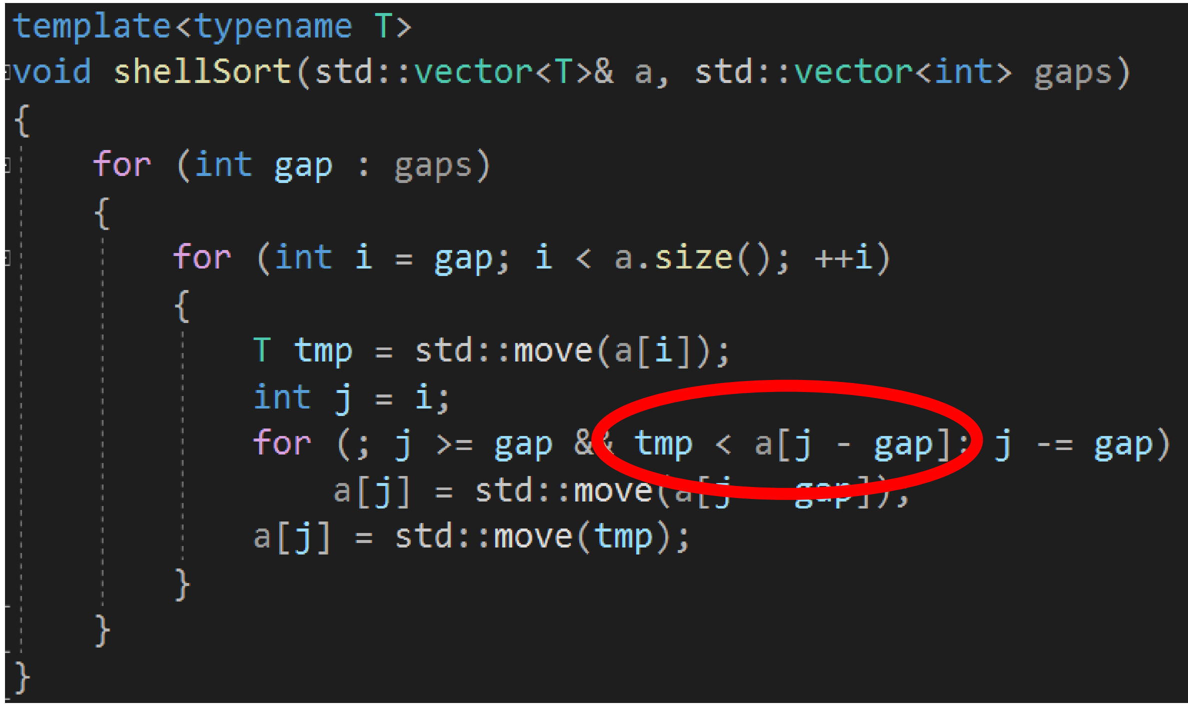
Lower bounds for sorting
and
Sorting in linear time

Lower bounds for sorting

- Have already seen several algorithms for sorting in $O(nlogn)$
    - Quicksort (average case), shell sort (average case), merge sort (worst case)
- What do all these algorithms have in common?
- They use *comparisons* to sort the elements:



```
bool operator<(const LargeTypeRaw& rhs) {
    return (size < rhs.get_size());
}
```



```
template<typename T>
void shellSort(std::vector<T>& a, std::vector<int> gaps)
{
    for (int gap : gaps)
    {
        for (int i = gap; i < a.size(); ++i)
        {
            T tmp = std::move(a[i]);
            int j = i;
            for (; j >= gap && tmp < a[j - gap]; j -= gap)
                a[j] = std::move(a[j - gap]);
            a[j] = std::move(tmp);
        }
    }
}
```

Lower bounds for sorting

- **Fact:** Any comparison sort must make $\Omega(nlogn)$ comparisons in the worst case to sort n elements
- ⇒ merge sort is an asymptotically optimal sorting algorithm
  - No comparison sort exists that is faster by more than a constant factor
- Example of non-comparison sort: trailmix sort
  - Smaller seeds at bottom, larger seeds at top

Lower bound for sorting

- In comparison sort, we use only comparisons between elements to gain information about ordering in a sequence $< \mathrm{a}_1, \mathrm{a}_2, \ldots, a_n >$
- Given two elements a_i and a_j, check one of: $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ or $a_i > a_j$
- **Simplifying assumptions:**
- Assume all elements are distinct (no need to check $a_i = a_j$)
- Also, comparisons $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$ and $a_i > a_j$ all yield identical information about the relative order of a_i and a_j
- Therefore assume all comparisons have form $a_i \leq a_j$

Decision tree model

- Comparison sorts can be viewed abstractly in the form of *decision trees*
- Decision tree: a full binary tree that represent the comparisons between elements performed by a particular sorting algorithm operating on an input of a given size
- Figure 8.1: decision tree corresponding to insertion sort operating on input sequence of 3 elements

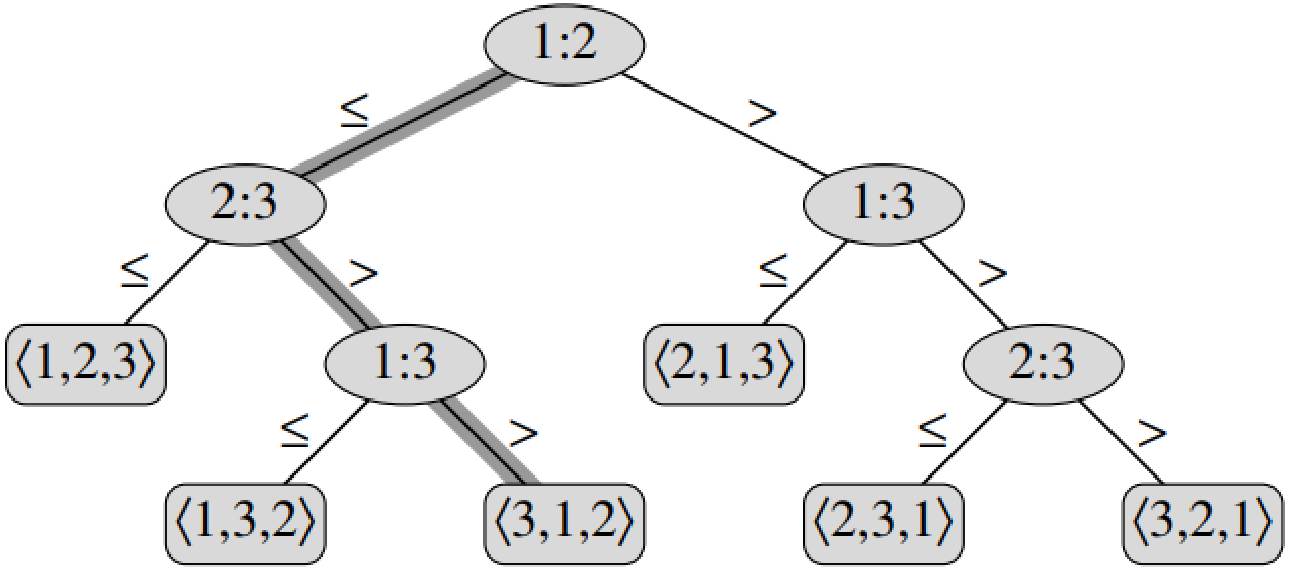
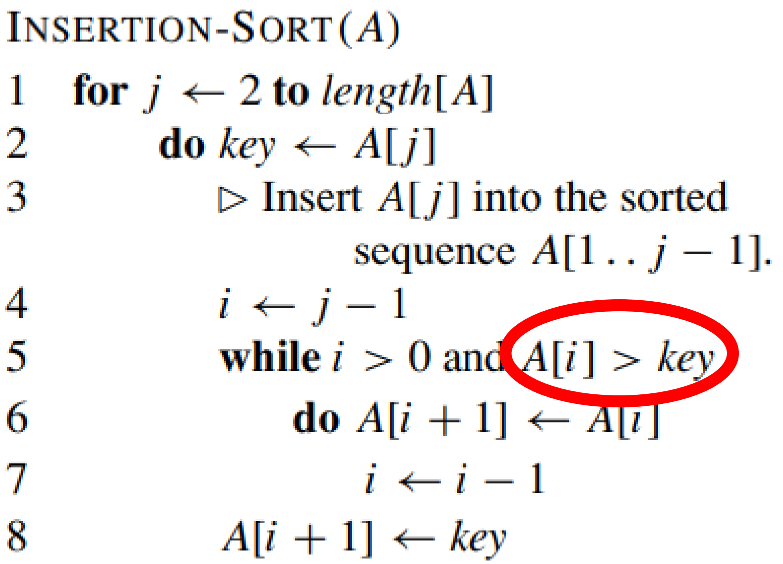
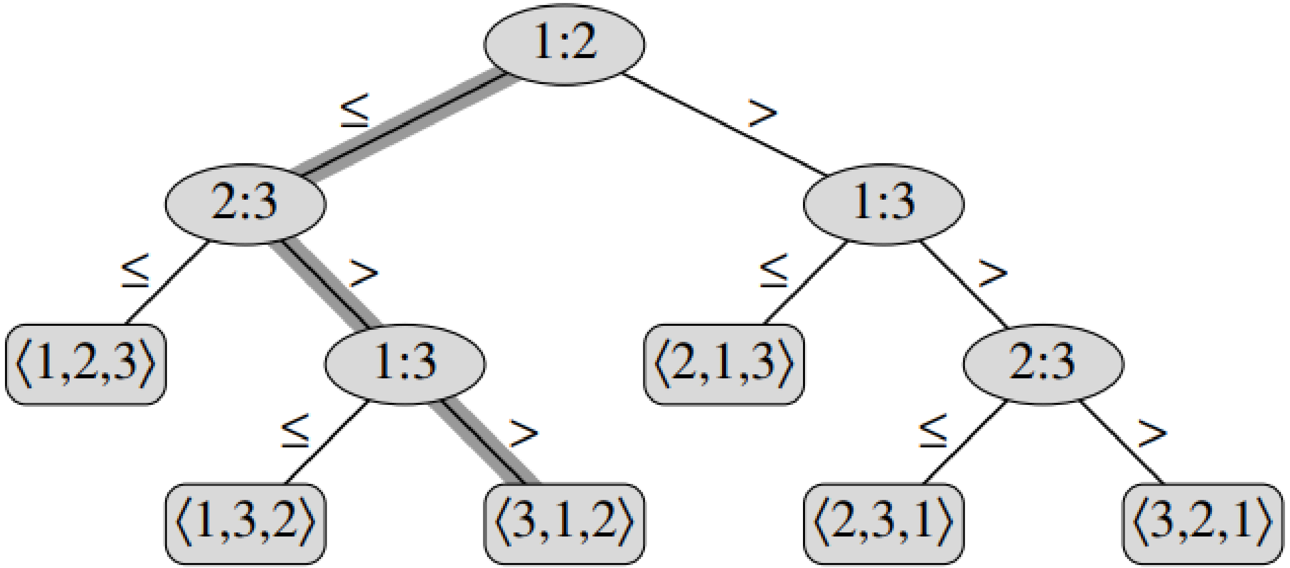


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i\!:\!j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Decision tree model



```
INSERTION-SORT($A$)
for $j \leftarrow 2$ to $length[A]$
    do $key \leftarrow A[j]$
       ▷ Insert $A[j]$ into the sorted
            sequence $A[1 \,.\,.\, j-1]$.
       $i \leftarrow j-1$
       while $i > 0$ and $A[i] > key$
           do $A[i+1] \leftarrow A[i]$
              $i \leftarrow i-1$
       $A[i+1] \leftarrow key$
```

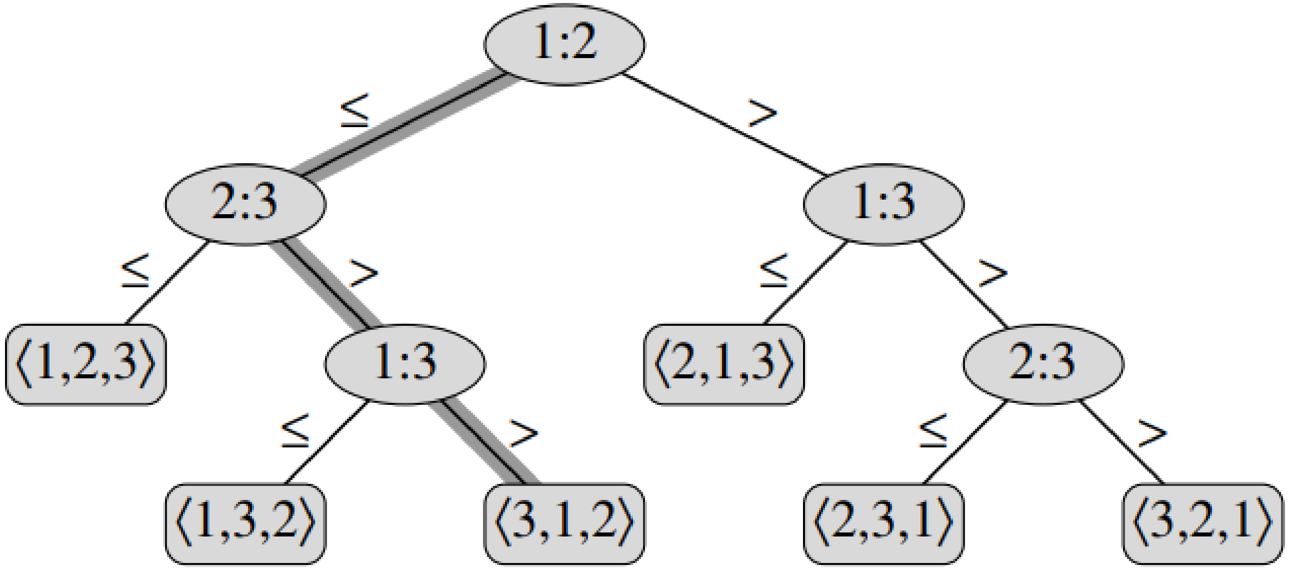


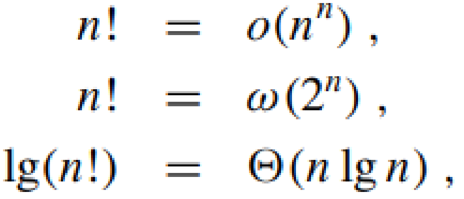
A: [a,b,c]
a=A[1], b=A[2], c=A[3]

Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Decision tree model

- Any correct sorting algorithm must be able to produce each permutation of its input
- Number of permutations for a sequence of length n is $n!$
- Therefore, the decision tree (which is a binary tree) corresponding to any input of size n must have at least $n!$ leaves





$$\begin{aligned} n! &= o(n^n) , \\ n! &= \omega(2^n) , \\ \lg(n!) &= \Theta(n \lg n) , \end{aligned}$$

Equation 3.18

proof

Proof

- **Theorem:** any comparison sort algorithm requires $\Omega(nlogn)$ comparisons in the worst case
- **Proof:** need to determine height of a decision tree in which each permutation appears as a reachable leaf.
- Consider decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements
- Because each of the $n!$ permutations of the input appears as some leaf, we have $n! < l$.
- Since a binary tree of height h has no more than 2^h leaves, we have
- $n! \leq l \leq 2^h \Rightarrow h \geq \log(n!) = \Omega(nlogn)$ (equation 3.18)
- **Corollary**: mergesort is an asymptotically optimal comparison sort

Counting sort

- **Counting sort** assumes each of the n input elements is an integer in range 0 to k, for some integer k
- When $k = O(n)$, counting sort runs in $\Theta(n)$ time
- Basic idea: determine, for each input element x, the number of elements less than x, and then place x in its position in the array
  - Example: if there are 17 elements less than x, x belongs at position 18
  - Need also to account for duplicates

Counting sort

```
COUNTING-SORT($A, B, k$)
 1  for $i \leftarrow 0$ to $k$
 2        do $C[i] \leftarrow 0$
 3  for $j \leftarrow 1$ to $length[A]$
 4        do $C[A[j]] \leftarrow C[A[j]] + 1$
 5  ▷ $C[i]$ now contains the number of elements equal to $i$.
 6  for $i \leftarrow 1$ to $k$
 7        do $C[i] \leftarrow C[i] + C[i - 1]$
 8  ▷ $C[i]$ now contains the number of elements less than or equal to $i$.
 9  for $j \leftarrow length[A]$ downto 1
10        do $B[C[A[j]]] \leftarrow A[j]$
11           $C[A[j]] \leftarrow C[A[j]] - 1$
```

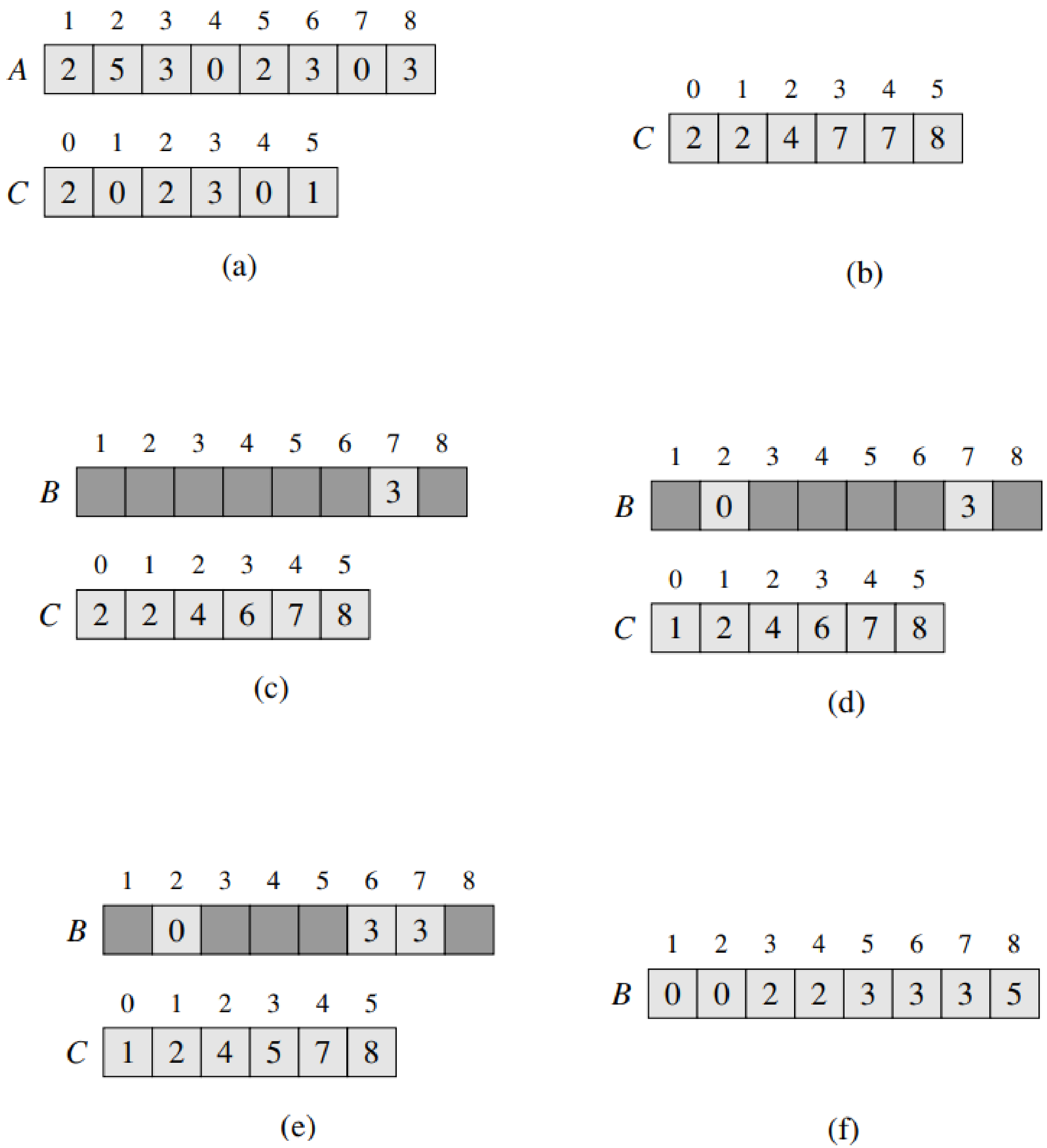
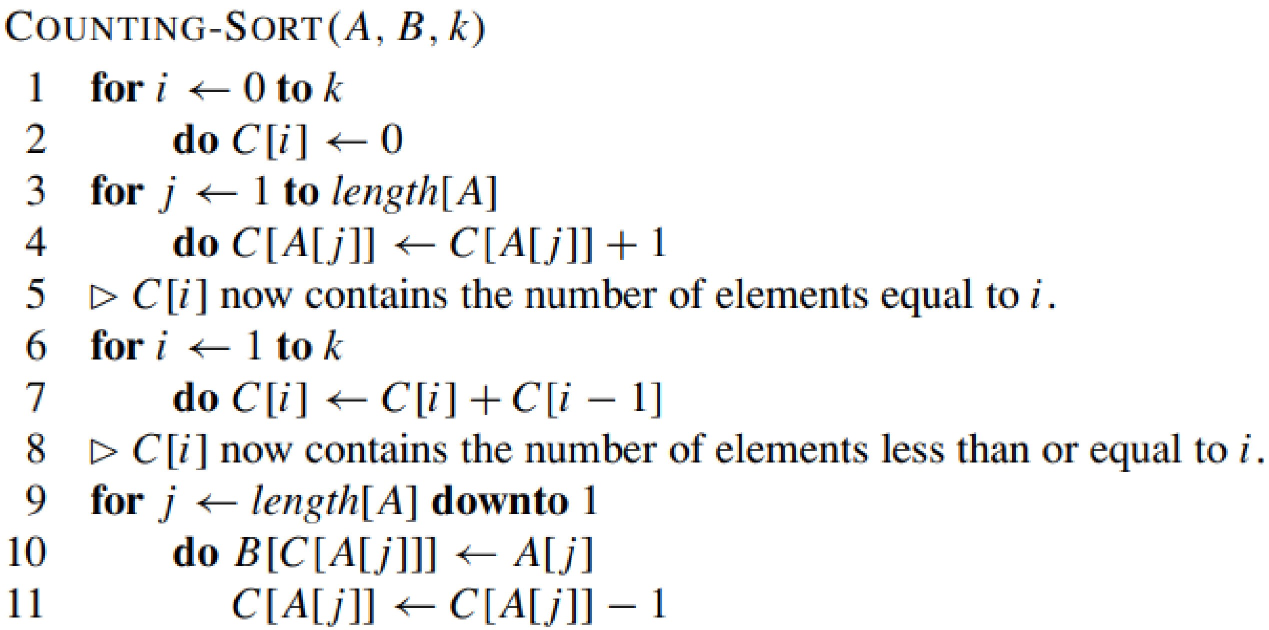


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting sort analysis



```
COUNTING-SORT($A$, $B$, $k$)
 1  for $i \leftarrow 0$ to $k$
 2       do $C[i] \leftarrow 0$
 3  for $j \leftarrow 1$ to $length[A]$
 4       do $C[A[j]] \leftarrow C[A[j]] + 1$
 5  ▷ $C[i]$ now contains the number of elements equal to $i$.
 6  for $i \leftarrow 1$ to $k$
 7       do $C[i] \leftarrow C[i] + C[i-1]$
 8  ▷ $C[i]$ now contains the number of elements less than or equal to $i$.
 9  for $j \leftarrow length[A]$ downto 1
10       do $B[C[A[j]]] \leftarrow A[j]$
11          $C[A[j]] \leftarrow C[A[j]] - 1$
```

- ‘for’ loop on lines 1-2 takes $\Theta(k)$
- ‘for’ loop on lines 3-4 takes $\Theta(n)$
- ‘for’ loop on lines 6-7 takes $\Theta(k)$
- ‘for’ loop on lines 9-11 takes $\Theta(n)$
- Thus, overall time is $\Theta(k + n)$
- If $k \leq O(n)$, counting sort running time is $\Theta(n)$.
  - This is faster than $\Theta(nlogn)$, made possible because no comparisons are performed by counting sort

Stable vs non-stable sorts

- **Stable** sorting algorithm: any sorting algorithm where numbers with the same value appear in the output array in the same order as they do in the input array
- Ties between two numbers are broken by the rule that whichever number appears first in the input array appears first in the output array
- Example: sorting strings digit-by-digit
- a) unsorted list
- b) after sorting on least-significant char (rightmost char)
- c) after sorting on middle char
- d) after sorting on most-significant char (leftmost char)
  - If sort was not stable, BAB and BBA could end up switched around, producing an incorrect result

a)
BAB
ABB
BBA

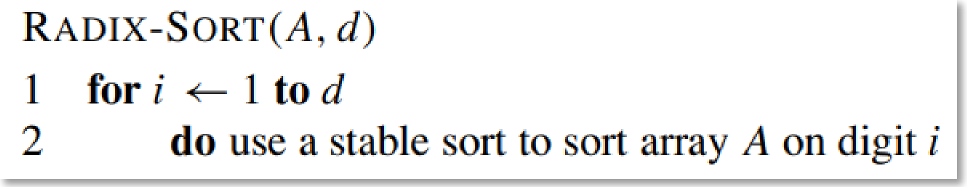
b)
BB**A**
AB**B**
BA**B**

c)
B**A**B
B**B**A
A**B**B

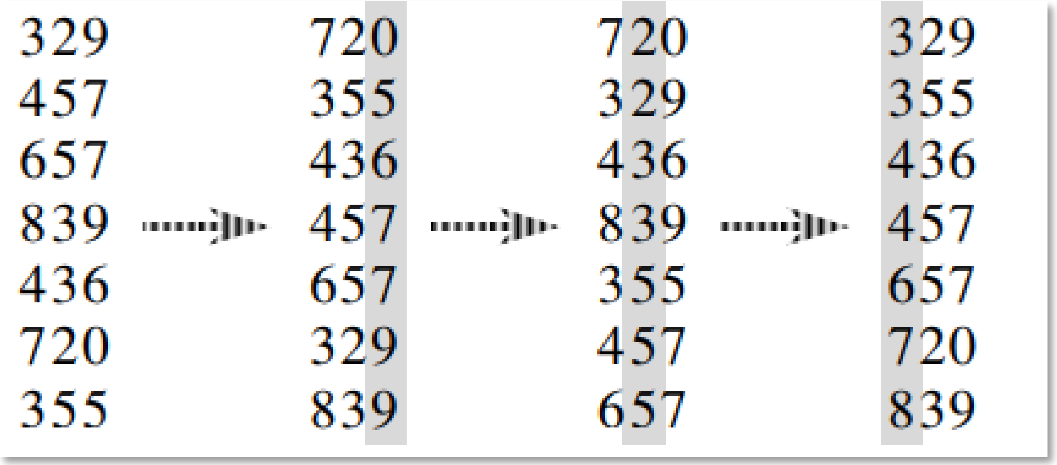
d)
ABB
BAB
BBA

Radix sort

- Radix sort simply calls a stable sort (such as counting sort) starting at the least-significant digit, all the way up to the most significant digit
- Allows to reduce the 'k' from counting sort



```
RADIX-SORT$(A, d)$
for $i \leftarrow 1$ to $d$
    do use a stable sort to sort array $A$ on digit $i$
```



329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.